#7 "3D Computer Graphics" by: Alan Watt (3rd)

附件一

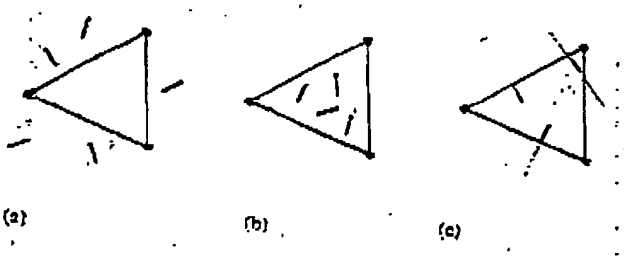


Figure 5.7
Clipping against a view volume – a routine polygon operation in the pipeline.
(a) Polygons outside the view volume are discarded.
(b) Polygons inside the view volume are retained.
(c) Polygons intersecting a boundary are clipped.

$$x_v = \pm \frac{hz_v}{d}$$

$$y_v = \pm \frac{hz_v}{d}$$

Clipping against the view volume (Figure 5.7) can now be carried out using polygon plane intersection calculations given in Section 1.4.3. This illustrates the principle of clipping, but the calculations involved are more efficiently carried out in three-dimensional screen space as we shall see.

Three-dimensional screen space

The final three-dimensional space in our pipeline we call three-dimensional screen space. In this space we carry out (practical) clipping against the view volume and the rendering processes that we will describe later. Three-dimensional screen space is used because it simplifies both clipping and hidden surface removal – the classic hidden surface removal algorithm being the Z-buffer algorithm which operates by comparing the depth values associated with different objects that project onto the same pixel. Also in this space there is a final transformation to two-dimensional view plane coordinates – sometimes called the perspective divide. (The terms 'screen' and 'view plane' mean slightly different things. Strictly speaking screen coordinates are derived from view plane coordinates by a device-dependent transformation.)

Because the viewing surface in computer graphics is deemed to be flat we consider the class of projections known as planar geometric projections. Two basic projections, perspective and parallel, are now described. These projections and the difference in their nature is illustrated in Figure 5.8.

A perspective projection is the more popular or common choice in computer graphics because it incorporates foreshortening. In a perspective projection relative dimensions are not preserved, and a distant line is displayed smaller than a nearer line of the same length (Figure 5.9). This effect enables human beings to perceive depth in a two-dimensional photograph or a stylization of three-dimensional reality. A perspective projection is characterized by a point known

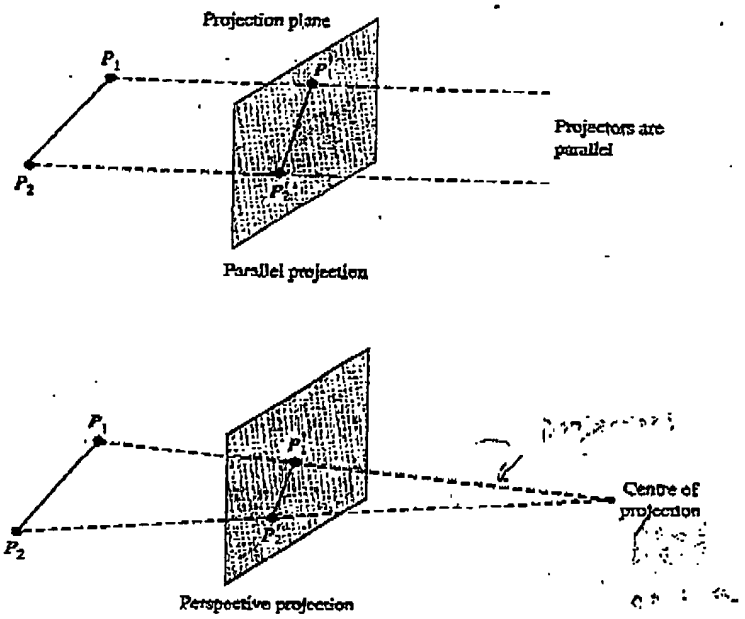


Figure 5.8
Two points projected onto a plane using parallel and perspective projections.

as the centre of projection and the projection of three-dimensional points onto the view plane is the intersection of the lines from each point to the centre of projection. These lines are called projectors.

Figure 5.10 shows how a perspective projection is derived. Point P (x_w, y_w, z_w) is a three-dimensional point in the view coordinate system. This point is to be projected onto a view plane normal to the z_w axis and positioned at distance d from the origin of this system. Point P' is the projection of this point in the view plane and has two-dimensional coordinates (x_s, y_s) in a view plane coordinate system with the origin at the intersection of the z_v axis and the view plane.

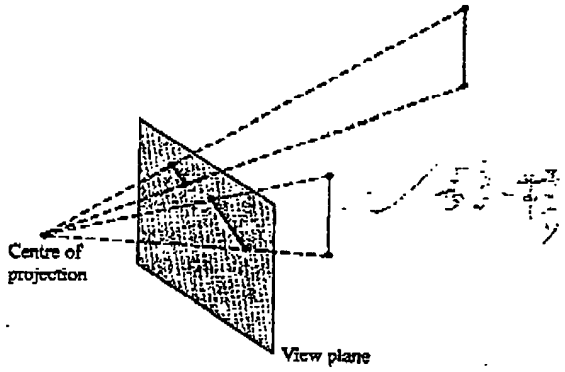
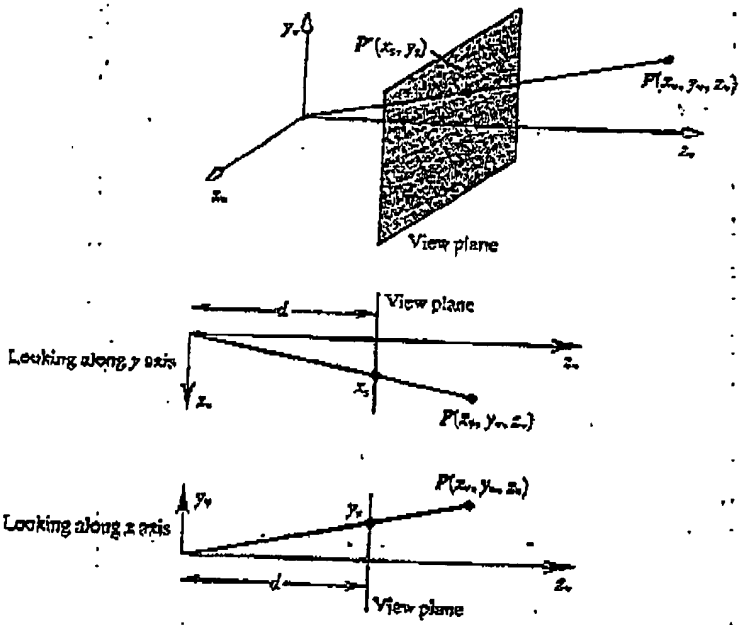


Figure 5.9
In a perspective projection a distant line is displayed smaller than a nearer line the same length.

Figure 5.10
Deriving a perspective transformation.



Similar triangles give:

$$\frac{x_s}{d} = \frac{x_v}{z_v} \qquad \frac{y_s}{d} = \frac{y_v}{z_v}$$

To express this non-linear transformation as a 4×4 matrix we can consider it in two parts – a linear part followed by a non-linear part. Using homogeneous coordinates we have:

$$X = x_v$$
$$Y = y_v$$
$$Z = z_v$$
$$w = z_v/d$$

We can now write:

$$\begin{bmatrix} X \\ Y \\ Z \\ w \end{bmatrix} = T_{pers} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix}$$

where:

$$T_{pers} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

following this with the perspective divide, we have:

$$x_s = X/w$$
$$y_s = Y/w$$
$$z_s = Z/w$$

In a parallel projection, if the view plane is normal to the direction of projection then the projection is orthographic and we have:

$$x_s = x_v \quad y_s = y_v \quad z_v = 0$$

Expressed as a matrix:

$$T_{ort} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5.2.4 View volume and depth

We now consider extending the above simple transformations to include the simplified view volume introduced in Figure 5.6. We discuss in more detail the transformation of the third component of screen space, namely z_s – ignored so far because the derivation of this transformation is somewhat subtle. Now, the bulk of the computation involved in rendering an image takes place in screen space. In screen space polygons are clipped against scan lines and pixels, and hidden surface calculations are performed on these clipped fragments. In order to perform hidden surface calculations (in the Z-buffer algorithm) depth information has to be generated on arbitrary points within the polygon. In practical terms this means, given a line and plane in screen space, being able to intersect the line with the plane, and to interpolate the depth of this intersection point, lying on the line, from the depth of the two end points. This is only a meaningful operation in screen space providing that in moving from eye space to screen space, lines transform into lines and planes transform into planes. It can be shown (Newman and Sproull 1973) that these conditions are satisfied provided the transformation of z takes the form:

$$z_s = A + B/z_v$$

where A and B are constants. These constants are determined from the following constraints:

(1) Choosing $B < 0$ so that as z_v increases then so does z_s. This preserves our intuitive Euclidean notion of depth. If one point is behind another, then it will have a larger z_v value, if $B < 0$ it will also have a larger z_s value.

(2) An important practical consideration concerning depth is the accuracy to which we store its value. To ensure this is as high as possible we normalize the range of z_s values so that the range $z_v \in [d, f]$ maps into the range $z_s \in [0, 1]$.

Considering the view volume in Figure 5.6, the full perspective transformation is given by:

$$x_s = d\frac{x_v}{hz_v}$$

$$y_s = d\frac{y_v}{hz_v}$$

$$z_s = \frac{f(1-d/z_v)}{(f-d)}$$

where the additional constant, h, appearing in the transformation for x_s and y_s ensures that these values fall in the range $[-1, 1]$ over the square screen. Adopting a similar manipulation to Section 5.2.3, we have:

$$X = \frac{d}{h}x_v$$

$$Y = \frac{d}{h}y_v$$

$$Z = \frac{fz_v}{f-d} - \frac{df}{f-d}$$

$$w = z_v$$

giving:

$$\begin{bmatrix} X \\ Y \\ Z \\ w \end{bmatrix} = T_{persp}\begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix}$$

where:

$$T_{persp} = \begin{bmatrix} d/h & 0 & 0 & 0 \\ 0 & d/h & 0 & 0 \\ 0 & 0 & f/(f-d) & -df/(f-d) \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad [5.1]$$

We can now express the overall transformation from world space to screen space as a single transformation obtained by concatenating the view and perspective transformation giving:

$$\begin{bmatrix} X \\ Y \\ Z \\ w \end{bmatrix} = T_{persp}T_{view}\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

It is instructive to consider the relationship between z_v and z_s a little more closely; although as we have seen by construction, they both provide a measure of the depth of a point, interpolating along a line in eye space is not the same as interpolating this line in screen space. Figure 5.11 illustrates this point. Equal

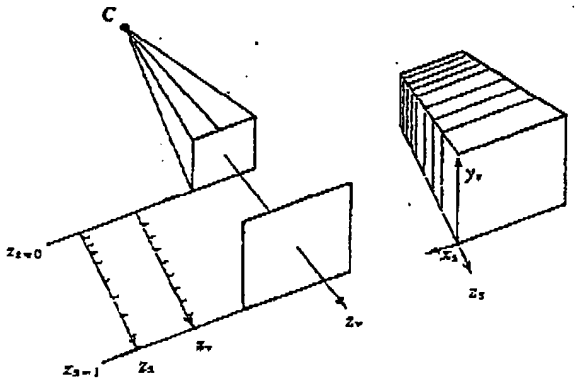


Figure 5.11
Illustrating the distortion in three-dimensional screen space due to the z_v to z_s transformation.

intervals in z_v are compared with the corresponding intervals in z_s. As z_v approaches the far clipping plane, z_s approaches 1 more rapidly. Thus, objects in screen space get pushed and distorted towards the back of the viewing frustum. This difference can lead to errors when interpolating quantities, other than position, in screen space.

In spite of this difficulty, by its very construction screen space is eminently suited to perform the hidden surface calculation. All rays passing through the view point are now parallel to the z_s axis because the centre of projection has been moved to negative infinity along the z_s axis. This can be seen by putting $z_v = 0$ into the above equation giving $z_s = -\infty$. Making those rays that hit the eye parallel, in screen space, means that hidden surface calculation need only be carried out on those points that have the same (x_s, y_s) coordinates. The test reduces to a simple comparison between z_s values to tell if a point is in front of another.

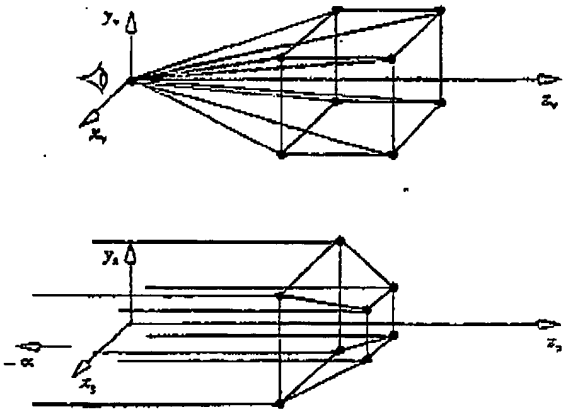


Figure 5.12
Transformation of box and light rays from eye space to screen space.

The transformation of a box with one side parallel to the image plane is shown in Figure 5.12. Here rays from the vertices of the box to the view point become parallel in three-dimensional screen space.

The overall precision required for the screen depth is a function of scene complexity. For most scenes 8 bits is insufficient and 16 bits usually suffices. The effects of insufficient precision is easily seen when, for example, a Z-buffer algorithm is used in conjunction with two intersecting objects. If the objects exhibit a curve where they intersect, this will produce aliasing artefacts of increasing severity as the precision of the screen depth is reduced.

Now return to the problem of clipping. It is easily seen from Figure 5.12 that in the homogeneous coordinate representation of screen space the sides of the view volume are parallel. This means that clipping calculations reduce to limit comparisons – we no longer have to substitute points into plane equations. The clipping operations must be performed on the homogeneous coordinates before the perspective divide, and translating the definition of the viewing frustum into homogeneous coordinates gives us the clipping limits:

$$-w \le x \le w$$
$$-w \le y \le w$$
$$0 \le z \le w$$

It is instructive to also consider the view space to eye space transformation by splitting Equation 5.1 into a product:

$$\mathbf{T}_{pers} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & f/(f-d) & -df/(f-d) \\ 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} d/h & 0 & 0 & 0 \\ 0 & d/h & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \mathbf{T}_{pers2}\mathbf{T}_{pers1}$$

This enables a useful visualization of the process. The first matrix is a scaling (d/h) in x and y. This effectively converts the view volume from a truncated pyramid with sides sloping at an angle determined by h/d into a regular pyramid with sides sloping at 45° (Figure 5.13). For example, point:

$(0, h, d, 1)$ transforms to $(0, d, d, 1)$

and point:

$(0, -h, d, 1)$ transforms to $(0, -d, d, 1)$

The second transformation maps the regular pyramid into a box. The near plane maps into the (x, y) plane and the far plane is mapped into $z=1$. For example, point:

$(0, d, d, 1)$ transforms to $(0, d, 0, d)$

which is equivalent to $(0, 1, 0, 1)$.

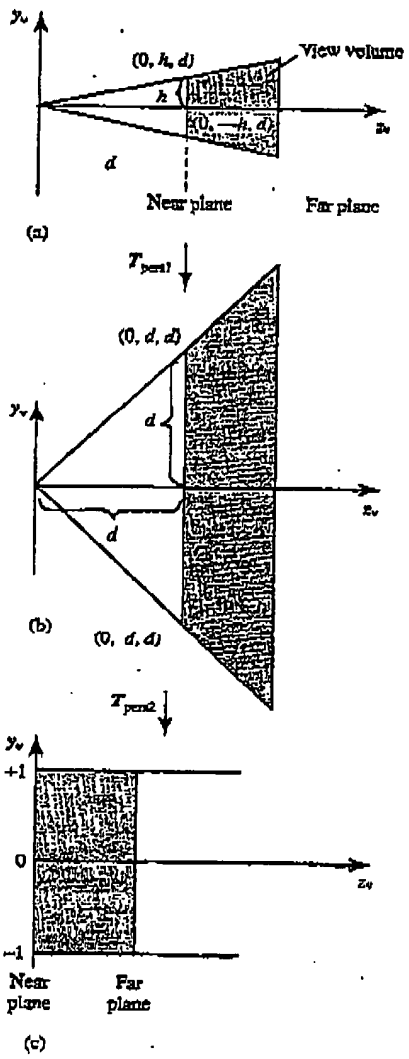


Figure 5.13
Transformation of the view volume into a canonical view volume (a box) using two matrix transformations.

5.3 Advanced viewing systems (PHIGS and GKS)

The viewing systems defined by the graphics standards PHIGS and GKS are far more general and more difficult to implement and understand than the system just described and so this section is very much optional reading. An unfortunate aspect of the standards viewing systems is that because they afford such generality they are hopelessly cumbersome and difficult to interface with. Even if a subset of parameters is used the default values for the unused parameters have

Finally, it is worth pointing out that it is possible to achieve a hybrid of these two methods. Often it is possible to split a scene up into a number of unconnected objects. If the scene is rendered on an object-by-object basis, using scan line ordering within each object, then the advantage of shared information is realized within each object, and there is no upper limit on scene complexity, only on the complexity of each individual object.

6.6

Hidden surface removal

The major hidden surface removal algorithms are described in most computer graphics textbooks and are classified in an early, but still highly relevant, paper by Sutherland *et al.* (1974) entitled 'A characterization of ten hidden-surface algorithms'. In this paper algorithms are characterized as to whether they operate primarily in object space or image (screen) space and the different uses of 'coherence' that the algorithms employ. Coherence is a term used to describe the process where geometrical units, such as areas or scan line segments, instead of single points, are operated on by the hidden surface removal algorithm.

There are two popular approaches to hidden surface removal. These are scan-line-based systems and Z-buffer-based systems. Other approaches to hidden surface removal such as area subdivision (Warnock 1969), or depth list schemes (Newell *et al.* 1972) are not particularly popular or are reserved for special-purpose applications such as flight simulation.

6.1

The Z-buffer algorithm

The Z-buffer algorithm, developed by Catmull (1975), is as ubiquitous in computer graphics as the Phong reflection model and interpolator, and the combination of these represents the most popular rendering option. Using Sutherland's classification scheme (Sutherland *et al.* 1974), it is an algorithm that operates in image, that is, screen space.

Pixels in the interior of a polygon are shaded, using an incremental shading scheme, and their depth is evaluated by interpolation from the z values of the polygon vertices after the viewing transformation has been applied. The equations in Section 1.5 are used to interpolate the depth values.

The Z-buffer algorithm is equivalent, for each point (x_s, y_s) to a search through the associated z values of each interior polygon point, to find that point with the minimum z value. This search is conveniently implemented by using a Z-buffer, that holds for a current point (x, y) the smallest z value so far encountered. During the processing of a polygon we either write the intensity of a point (x, y) into the frame buffer, or not, depending on whether the depth z, of the current point, is less than the depth so far encountered as recorded in the Z-buffer.

One of the major advantages of the Z-buffer is that it is independent of object representation form. Although we see it used most often in the context of poly-

gon mesh rendering, it can be used with any representation – all that is required is the ability to calculate a z value for each point on the surface of an object. It can be used with CSG objects and separately rendered objects can be merged into a multiple object scene using Z-buffer information on each object. These aspects are examined shortly.

The overwhelming advantage of the Z-buffer algorithm is its simplicity of implementation. Its main disadvantage is the amount of memory required for the Z-buffer. The size of the Z-buffer depends on the accuracy to which the depth value of each point (x, y) is to be stored; which is a function of scene complexity. Between 20 and 32 bits is usually deemed sufficient and the scene has to be scaled to this fixed range of z so that accuracy within the scene is maximized. Recall in the previous chapter that we discussed the compression of z_v values. This means that a pair of distinct points with different z_v values can map into identical z_s values. Note that for frame buffers with less than 24 bits per pixel, say, the Z-buffer will in fact be larger than the frame buffer. In the past, Z-buffers have tended to be part of the main memory of the host processor, but now graphics terminals are available with dedicated Z-buffers and this represents the best solution.

The memory problem can be alleviated by dividing the Z-buffer into strips or partitions in screen space. The price paid for this is multiple passes through the geometric part of the renderer. Polygons are fetched from the database and rendered if their projection falls within the Z-buffer partition in screen space.

An interesting use of the Z-buffer is suggested by Foley *et al.* (1989). This involves rendering selected objects but leaving the Z-buffer contents unmodified by such objects. The idea can be applied to interaction where a three-dimensional cursor object can be moved about in a scene. The cursor is the selected object, and when it is rendered in its current position, the Z-buffer is not written to. Nevertheless the Z-buffer is used to perform hidden surface removal on the object and will move about the scene obscuring some objects and being obscured by others.

6.6.2 Z-buffer and CSG representation

The Z-buffer algorithm can be used to advantage in rendering CSG objects. As you will recall from Section 4.3, which describes a ray tracing algorithm for rendering such objects, rendering involves calculating a boundary representation of a complex object that is made up of primitive objects combined with Boolean operators and described or represented by a construction tree.

The problem with the ray tracing method is expense. A normal recursive ray tracer is a method that finds intersections between a ray of arbitrary direction and objects in the scene. This model operates recursively to any depth to evaluate specular interaction. However, with CSG objects, all rays are parallel and we are only interested in the first hit, so in this respect ray tracing is inappropriate and a Z-buffer approach is easier to implement and less expensive (Rossignac and